# Programmazione Funzionale I

(Francesco Galgani - www.galgani.it)

# Indice

CAPITOLO 1

# Introduzione

*Nota* *Gli esempi e gli esercizi a corredo delle parti teoriche sono parzialmente tratti dal libro "Introduzione alla Programmazione Funzionale"* [**5**].

## 1.1. La programmazione funzionale

tratto da: *"Programmazione Funzionale - Numero 1" di V. Ciancia e G. Belmonte* [1]

*« In un'epoca di comunicazione di massa, di diffusione di massa dell'informatica, di diffusione di massa della cultura, è quasi implicito che anche la programmazione sia passata da scienza a cultura generale e sia divenuta alla portata di tutti.*

*Linguaggi di programmazione di ogni genere sono largamente diffusi ed usati; ciò nonostante, invece di avere un forte progresso della qualità dei programmi, abbiamo avuto dagli ultimi dieci anni uno spaventoso incremento degli errori di programmazione. La prima causa è senz'altro la maggiore complessità dei programmi, ma fra le cause maggiori vi è anche il fatto che in programmazione si usano strumenti piuttosto antichi, che non fanno pieno uso di tutte le scoperte fatte dalla scienza informatica negli ultimi decenni.*

*L'unico metodo efficace per affermare che un programma è esente da errori è la dimostrazione di correttezza rispetto alle specifiche, ma questa è indecidibile, quindi non può essere automatizzata, ed è molto complesso riuscire a dimostrare, anche a mano, la correttezza di un lungo programma.*

*Si possono invece limitare, ma non evitare del tutto, gli errori di programmazione fornendo una dimostrazione di correttezza parziale, cioè relativa solo a parte di una specifica. Una dimostrazione di correttezza parziale molto usata (anche inconsciamente) è il controllo di tipi statico: si specificano nel programma i tipi (che sono una versione meno vincolante rispetto alle specifiche) e il compilatore riesce a controllare che i tipi effettivi siano conformi a quelli dichiarati.*

*Molti dei linguaggi diffusi, come il C e il C++ per citare i più famosi, hanno un sistema di tipi debole, nel senso che è possibile violare il controllo di tipi, ad esempio convertendo la rappresentazione di un valore da un tipo all'altro usando i puntatori.*

*Altri linguaggi, meno diffusi perché più complessi da usare e meno immediati, hanno un controllo di tipi forte. Fra questi ce ne sono alcuni, come OCaml e Haskell, che fanno parte della categoria dei linguaggi funzionali; questa categoria è interessante perché propone un'astrazione, quello di funzione come valore, che consente di esprimere un concetto difficilmente simulabile in altri linguaggi: quello di funzioni che manipolano altre funzioni. Questo concetto, che difficilmente si è portati ad adoperare se non lo si conosce, è molto usato in matematica ed è un mezzo efficace di formalizzazione della soluzione di molti problemi. [...] »*

[1] http://xoomer.virgilio.it/ubrcianc/EPF/numero_1/numero_1.pdf

## 1.2. Tratti salienti della programmazione funzionale

La programmazione funzionale rappresenta un modo completamente diverso di affrontare i problemi, rispetto alla tradizionale programmazione imperativa basata sui comandi.

ML è l'acronimo di "Meta Language", che identifica un linguaggio di programmazione funzionale general-purpose sviluppato negli anni '70, allo scopo di realizzare un meta-linguaggio per programmare un dimostratore di teoremi in grado di seguire diverse strategie di dimostrazione.
Vedi: *http://it.wikipedia.org/wiki/ML* e *http://it.wikipedia.org/wiki/Ocaml*

Objective Caml, *http://caml.inria.fr/*, creato nel 1996, è un avanzato linguaggio di programmazione appartenente alla famiglia dei linguaggi ML. Alcune caratteristiche di questi linguaggi possono essere così sintetizzate:

(1) L'attenzione del programmatore tende a focalizzarsi sul "che cosa" piuttosto che sul "come": un programma funzionale dovrebbe descrivere il problema da risolvere piuttosto che indicare il meccanismo di soluzione.
(2) La programmazione funzionale richiede uno stile di pensiero decisamente astratto, fortemente ispirato ai principi della matematica.
(3) Un programma funzionale è costituito dalla definizione di un insieme di funzioni, che possono richiamarsi l'una con l'altra. Le funzioni di base sono trattate come "oggetti di prima classe", a partire dai quali è possibile definire "funzioni di ordine superiore", cioè funzioni che prendono funzioni come argomento o che restituiscono funzioni in uscita. In altre parole:
  (a) una funzione può essere una componente di una struttura dati;
  (b) una funzione può essere un argomento di un'altra funzione;
  (c) una funzione può essere un valore restituito da un'altra funzione.
(4) I costrutti di base sono espressioni, non comandi.
(5) Le espressioni sono costruite a partire da costanti e variabili (le quali a loro volta sono espressioni), mediante l'applicazione e la composizione di operazioni. La programmazione funzionale è quindi una *"programmazione orientata alle espressioni"*, nella quale la modalità fondamentale di calcolo è la "valutazione di espressioni", che consiste nel calcolare il valore di una data espressione, semplificandola fin dove possibile.
(6) Il codice di un programma funzionale è solitamente più conciso del corrispondente codice procedurale ed anche più affidabile.
(7) Non esistono strutture di controllo predefinite per la realizzazione di cicli `for` o `while`: il principale meccanismo di controllo è la ricorsione *(vedi 1.5.3 e 1.7)*.
(8) La programmazione imperativa si basa sul $\lambda$-calcolo (lambda-calcolo), sviluppato per analizzare formalmente le definizioni di funzioni, le loro applicazioni e i fenomeni di ricorsione.
(9) L'ambiente di valutazione delle espressioni, cioè la collezione di legami tra variabili e valori, viene gestito come uno *stack*: ogni nuova dichiarazione aggiunge un legame in cima alla pila, senza alterare i legami preesistenti. Il modulo Pervasives, che sta alla base dell'ambiente di valutazione, contiene le definizioni di tutte le variabili e funzioni predefinite in Ocaml.
(10) Non esiste l'assegnamento di valori alle variabili *(vedi 1.4)*.

### 1.3. Installazione e utilizzo di Ocaml

Il sito di riferimento è:

- The Caml Language
  *http://caml.inria.fr/*

nel quale è possibile scaricare distribuzioni di Ocaml per Linux, MacOS e Windows. Da notare, comunque, che l'installazione in ambiente Linux è generalmente possibile e semplificata grazie ai pacchetti precompilati per la propria distribuzione e disponibili nei relativi repository. Su Debian e derivati è sufficiente il seguente comando da root:

```
# apt-get install ocaml-core
```

Per aprire l'interprete dei comandi, si digiti `ocaml`. L'output sarà il seguente:

```
utente@ubuntu:~$ ocaml
        Objective Caml version 3.08.3
#
```

Nel prompt di Ocaml, contraddistinto dal simbolo di cancelletto, è possibile inserire espressioni (su uno o più righi), le quali saranno subito valutate dall'interprete, che ne stamperà il tipo (dedotto automaticamente, vedi 1.6.1) e il valore. Le espressioni terminano con due "punto e virgola" consecutivi. Ad esempio:

```
# 3 + 3;;
- : int = 6
```

Per maggiore comodità, è possibile memorizzare il codice in un file di testo, con l'editor preferito, ed eseguirlo con la direttiva:

```
# #use "nome_del_file";;
```

oppure integrare l'interprete in Emacs, come descritto nella guida:

- Using Ocaml on Linux
  *http://www.cis.ksu.edu/~ab/Courses/505/spr06/docs/ocaml.pdf*

### 1.4. Dichiarazioni di variabili

Una variabile viene introdotta nell'ambiente legandola ad un valore *(value binding)*, per mezzo della sintassi:

```
let Variabile = Espressione
```

Ocaml valuta prima la parte destra dell'equazione e poi ne lega il valore alla variabile specificata nella parte sinistra. I nomi delle variabili devono essere scritti minuscolo.

```
# let n = 2 + 3;;
val n : int = 5
```

Se la dichiarazione di una variabile o di una funzione contiene il riferimento ad una variabile `x`, ogni volta che tale variabile o funzione viene valutata è utilizzato il valore che `x` aveva nel momento in cui è stata introdotta la dichiarazione. Ad esempio, nel codice seguente la seconda dichiarazione di `x` crea un nuovo binding che maschera il primo, ma non intacca il valore di `y`. La parola chiave `and` effettua il binding in parallelo.

```
# let x = 17;;
val x : int = 17
# let y = x;;
val y : int = 17
# let x = true;;
val x : bool = true
# y;;
- : int = 17
# let x = false and y = x;;
val x : bool = false
val y : bool = true
```

Il valore di una variabile è definito dal `let` più recente.

## 1.5. Dichiarazioni di funzioni

Un programma in Ocaml è essenzialmente un insieme di definizioni di funzioni, espresse nella forma:

```
function Argomento -> Espressione
```

che possono essere dichiarate con una delle due sintassi:

```
let [rec] Nome (Arg1, Arg2, ..., Argn) = Espressione;;
let [rec] Nome = function (Arg1, Arg2, ..., Argn) -> Espressione;;
```

dove `rec` è una parola chiave da inserire solo nel caso di funzioni ricorsive, *`Nome`* è il nome della funzione e *`Arg1`* del primo argomento[2] (tutti i nomi devono essere scritti minuscoli). I tipi delle funzioni hanno la forma *`tipo1 -> tipo2`*, dove *`tipo1`* è il tipo degli argomenti della funzione e *`tipo2`* è quello dei valori restituiti.

```
# let doppio = function x -> 2*x;;
val doppio : int -> int = <fun>
# let triplo x = 3*x;;
val triplo : int -> int = <fun>
# doppio 4;;
- : int = 8
# triplo 4;;
- : int = 12
```

[2] *Formalmente le funzioni hanno un solo argomento. Più parametri sono passati usando una tupla* (vedi 1.6.7) *oppure mediante currificazione* (vedi 1.5.1).

**1.5.1. La curryficazione di funzioni (currying).** La curryficazione (così chiamata in onore del logico Haskell Curry[3]) consiste nel trasformare una funzione che prende più parametri attraverso una tupla in un'altra che può ricevere un sottoinsieme dei parametri della funzione originaria e restituire una nuova funzione, la quale può a sua volta prendere i parametri rimanenti e restituire il risultato.

Ad esempio, prendendo la funzione di due variabili:

```
function (y,x) -> y*x
```

e fissando `y=2`, si ottiene la funzione in una sola variabile:

```
function (x) -> 2*x.
```

Nella teoria dell'Informatica, la curryficazione fornisce un metodo per studiare funzioni con più argomenti attraverso modelli teorici molto semplici, come nel caso del lambda-calcolo, nel quale le funzioni prendono soltanto un singolo argomento.

*La motivazione pratica della curryficazione va ricercata nella possibilità di ottenere funzioni da funzioni*; per esempio, molti linguaggi hanno una funzione di incremento, facilmente implementabile tramite curryficazione della somma:

```
# (* funzione di partenza *)
  let somma (x,y) = x + y;;
val somma : int * int -> int = <fun>
# (* funzione curryficata *)
  let somma x y = x + y;;
val somma : int -> int -> int = <fun>
# (* fornisco il primo argomento alla funzione curryficata *)
  somma 1;;
- : int -> int = <fun>
# (* uso la funzione currifycata per definire una nuova funzione *)
  let incr = somma 1;;
val incr : int -> int = <fun>
# incr(5);;
- : int = 6
```

Il tipo di una funzione curryficata deve esser letto associando a destra: `int->int->int` equivale a `int->(int->int)`, quindi la funzione prende in input un intero e restituisce una funzione da interi a interi.

Ocaml dispone di alcune funzioni curryficate già predefinite nel modulo Pervasives, come le funzioni algebriche di base, che possono essere richiamate interponendo i relativi operatori algebrici tra parentesi tonde, e quelle di massimo e minimo:

```
# (+);;
- : int -> int -> int = <fun>
# (+.);;
- : float -> float -> float = <fun>
# (+.) 3.4 6.4;;
- : float = 9.8
# max;;
- : 'a -> 'a -> 'a = <fun>
# min;;
```

[3]*http://www-history.mcs.st-andrews.ac.uk/history/Mathematicians/Curry.html*

```
- : 'a -> 'a -> 'a = <fun>
```

Esempio di passaggio da versione non curryficata a curryficata e viceversa:

```
# (* CURRYFICA UNA FUNZIONE *)
   let curry f x y = f(x,y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# (* DECURRYFICA UNA FUNZIONE *)
  let uncurry f (x, y) = f x y;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

**1.5.2. Dichiarazioni locali.** Ocaml permette di avere dichiarazioni *locali,* cioè interne ad una data espressione, con la sintassi:

```
Dichiarazione in Espressione.
```

In questo caso, Ocaml estende provvisoriamente l'ambiente aggiungendo i legami dettati da *Dichiarazione* e, in tale ambiente esteso, valuta *Espressione*, per poi ripristinare l'ambiente preesistente.

```
# let n = 3 in if n<10 then 1 else 2;;
- : int = 1
# n;;
Unbound value n
```

Le dichiarazioni locali sono utili quando un identificatore ha un uso limitato e non ha significato al di fuori dell'espressione più ampia che lo utilizza. La definizione di funzioni utilizza spesso dichiarazioni locali, allo scopo di richiamare funzioni ausiliarie che non hanno significato autonomo.

**1.5.3. Il concetto di ricorsione.** Un algoritmo è *ricorsivo* se è espresso in termini di se stesso, generando un ciclo di chiamate che ha termine al verificarsi di una condizione particolare, detta *caso di base*, coincidente, in genere, con il presentarsi di particolari valori di input. Ad esempio, si consideri l'algoritmo di Euclide, il calcolo del fattoriale e dall'algoritmo delle Torri di Hanoi:

ALGORITHM 1.5.1. *calcolo del MCD con l'algoritmo di Euclide (vedi http://it.wikipedia.org/wiki/Algoritmo_di_Euclide).*

```
# let rec mcd (m,n) =
      if n=0 then m
      else mcd (n, m mod n);;
val mcd : int * int -> int = <fun>
```

ALGORITHM 1.5.2. *Calcolo del fattoriale di un intero positivo (vedi http://it.wikipedia.org/wiki/Ricorsione#Il_punto_di_partenza:_il_fattoriale).*

```
# let rec fatt (n) =
     if n=0 then 1
     else n * fatt (n-1);;
val fatt : int -> int = <fun>
```

ALGORITHM 1.5.3. *Le torri di Hanoi*
*Per una spiegazione del gioco e del suo algoritmo risolutivo, vedi: http://www.lia.deis.unibo.it/Courses/FondA-TLC/AslideHTML/10d.pdf*

## 1.6. I tipi

Vi sono espressioni complesse, che possono essere semplificate, ed espressioni semplici, i valori. Ogni espressione ha un suo valore e il processo di calcolo di tale valore è la *valutazione* dell'espressione. Un'espressione viene valutata in un *ambiente*, costituito da una collezione di legami variabile-valore.

- Un tipo è definito da un insieme di valori.
- Distinguiamo tra *tipi base* (i cui oggetti sono indivisibili: `unit`, `bool`, `int`, `float`, `char`) e *tipi composti* (i cui oggetti sono decomponibili in oggetti più semplici, di tipo base o anch'esso composto: `string`, `tuple`, `list`, `record`).
- Ocaml è *strettamente e fortemente tipato.* Questo significa che ogni espressione ha sempre e solo un tipo.
- Il tipo di una espressione è determinato a tempo di compilazione da un insieme di regole di *inferenza di tipo (vedi 1.6.1)*, che garantiscono che, se l'espressione ha un valore, esso sia del tipo assegnato all'espressione.
- Il *controllo dei tipi* consente di determinare se le espressioni sono usate con *coerenza* e di non autorizzare tentativi di calcolare espressioni senza senso, come ad esempio la sottrazione di un booleano da un intero o la somma di una stringa con un reale. Ad esempio:

```
# "ciao" + 3.14;;
This expression has type string but is here used with type int
```

### 1.6.1. Type inference (inferenza dei tipi).

In generale, l'inferenza dei tipi è l'abilità, a tempo di compilazione, di dedurre automaticamente (in modo parziale o completo) il tipo del valore restituito da un'espressione. Di solito il compilatore può dedurre il tipo di una variabile o di una funzione senza bisogno di esplicite dichiarazioni di tipo e, in molti casi, è addirittura possibile ometterle completamente.

Ocaml deduce il tipo di un'espressione da quello delle sottoespressioni, fino ad arrivare alle costanti, il cui tipo è dedotto dall'analizzatore lessicale. Ad esempio:

```
()        unit
true      booleano
42        intero
3.14159   reale
'a'       carattere
"ciao"    stringa
```

Questo non vuol dire che le dichiarazioni di tipo non sono possibili, semplicemente non sono necessarie.

### 1.6.2. Il polimorfismo.

I linguaggi della classe ML ammettono espressioni di tipo polimorfo, cioè oggetti con un tipo generico che può essere *istanziato* in diversi modi: istanziare un tipo polimorfo, quindi valido per qualsiasi tipo di dati, significa utilizzarlo con il tipo effettivamente richiesto dall'utente o dal contesto del programma. Ad esempio, se una funzione ha l'argomento polimorfo, allora posso passarle un intero, un reale, una tupla, una stringa o qualsiasi altro tipo.

L'*insieme di tutti i tipi possibili*, nella forma $t_1 \times t_2 \to t_1$, si rappresenta usando le lettere greche, in questo caso $\alpha \times \beta \to \alpha$. In Ocaml le lettere greche sono indicate premettendo l'apostrofo ad un carattere.

```
# let pippo (a,b) = a;;
val pippo : 'a * 'b -> 'a = <fun>
```

**1.6.3. Il tipo unit.** Ha un solo valore: `()`.

Si può pensare che `()` sia la tupla con zero elementi, utilizzata come argomento per "funzioni" senza argomenti o come valore di "funzioni" che di fatto corrispondono a procedure: il loro ruolo consiste negli effetti collaterali (come la stampa di una stringa) e non ha importanza il valore riportato.

```
# print_string;;
- : string -> unit = <fun>
```

**1.6.4. Booleani.** Il tipo `bool` è costituito unicamente dai due valori `false` e `true`. Le espressioni costruite mediante operatori di confronto sono di tipo `bool`.

Operazioni predefinite sui booleani: `not`, `&&` (oppure `&`), `||` (oppure `or`).

Nell'espressione condizionale `if` $E$ `then` $E_1$ `else` $E_2$, l'espressione $E$ deve avere un valore booleano; $E_1$ ed $E_2$ devono avere lo stesso tipo.

```
# let a = 3;;
val a : int = 3
# let b = 7;;
val b : int = 7
# if a>b then "maggiore" else "minore o uguale";;
- : string = "minore o uguale"
# if a>b then 1 else false;;
This expression has type bool but is here used with type int
```

**1.6.5. Interi e reali.** Gli interi (`int`) si scrivono nel modo abituale, invece i reali (`float`) sono sempre denotati dal punto decimale (indipendentemente dalla presenza o non di cifre decimali) oppure sono espressi in notazione esponenziale.

Operazioni predefinite sugli interi: `+ - * / mod succ() pred()`

```
# (* OPERAZIONI TRA INTERI *)
3 + 4;; (* somma *);;
- : int = 7
# 3 - 4;; (* sottrazione *);;
- : int = -1
# 3 * 4;; (* moltiplicazione *);;
- : int = 12
# 3 / 4;; (* divisione *);;
- : int = 0
# 3 mod 4;; (* modulo, cioè resto della divisione *);;
- : int = 3
# succ (3);; (* intero successivo *)
- : int = 4
# pred (3);; (* intero precedente *)
- : int = 2
```

Operazioni predefinite sui reali: `+.` `-.` `*.` `/.` `**` `sqrt()`

```
# (* OPERAZIONI TRA REALI *)
3. +. 4.;; (* somma *)
- : float = 7.
# 3. -. 4.;; (* sottrazione *)
- : float = -1.
# 3. *. 4.;; (* moltiplicazione *)
- : float = 12.
# 3. /. 4.;; (* divisione *)
- : float = 0.75
# 3. ** 4.;; (* elevamento a potenza *)
- : float = 81.
# sqrt (4.);; (* radice quadrata *)
- : float = 2.
# sqrt (3.);; (* radice quadrata *)
- : float = 1.73205080756887719
```

Da intero a reale e viceversa:

```
# int_of_float 3.9;;
- : int = 3
# float_of_int 3;;
- : float = 3.
```

### 1.6.6. Stringhe e caratteri.

- Le stringhe sono sequenze di caratteri.
- Il tipo `char` è distinto dal tipo `string` per l'uso, rispettivamente, degli apici singoli o doppi.
- Un carattere può essere indicato anche con una sequenza di escape o con il suo codice ASCII.
- Il carattere in posizione `n` in una stringa `s` è identificato da `s.[n]`, tenendo presente che il primo carattere è in posizione zero.
- L'operatore ^ permette di concatenare stringhe.

```
# 'n';;
- : char = 'n'
# "n";;
- : string = "n"
# '\110';; (* il backslash è seguito dal codice ASCII del carattere *)
- : char = 'n'
# '\n';; (* sequenza di escape per indicare un a capo *)
- : char = '\n'
# "naturale";;
- : string = "naturale"
# "\110"^"\097"^"turale";;
- : string = "naturale"
# "naturale".[2];;
- : char = 't'
```

Funzioni predefinite su stringhe e caratteri:

```
# "programmazione" ^ "funzionale";; (* concatenamento *)
- : string = "programmazionefunzionale"
# string_of_bool (true);; (* conversione da bool a string *)
- : string = "true"
# string_of_int (1);; (* conversione da int a string *)
- : string = "1"
# string_of_float (1.);; (* conversione da float a string *)
- : string = "1."
# int_of_string ("4");; (* conversione da string a int *)
- : int = 4
# float_of_string ("4.3");; (* conversione da string a float *)
- : float = 4.3
# String.length ("splendore");; (* numero di caratteri della stringa *)
- : int = 9
# Char.code ('a');; (* codice ASCII dell'argomento *)
- : int = 97
# Char.chr (98);; (* carattere corrispondente al codice fornito *)
- : char = 'b'
```

Su numeri, stringhe e caratteri sono definiti i seguenti operatori di confronto:

```
=  <  <=  >  >=  <>
# "ciao".[2] = 'a';;
- : bool = true
# 3 <> 4;;
- : bool = true
```

**1.6.7. Tuple.** Una tupla è una sequenza di lunghezza fissata di oggetti di tipo misto. Se $t_1$ e $t_2$ sono tipi, $t_1 \times t_2$ è il tipo delle coppie ordinate (tuple di due elementi) il cui primo elemento è di tipo $t_1$ ed il secondo il tipo $t_2$. Una coppia ordinata si scrive ($E_1$, $E_2$), dove $E_1$ ed $E_2$ sono espressioni. In Ocaml, il simbolo $\times$ è rappresentato con un asterisco.

```
# ("Lucia", 19);;
- : string * int = ("Lucia", 19)
# ((if 3>4 then true else false), sqrt(4.));;
- : bool * float = (false, 2.)
```

L'uguaglianza fra tuple viene valutata componente per componente. Non ha senso confrontare tuple di tipo diverso.

**1.6.8. Le liste.** Le liste, strutture dati fondamentali dei linguaggi funzionali, sono sequenze di oggetti dello stesso tipo, separati da `;` e racchiusi tra parentesi quadre.

```
# ["casa";"famiglia";"amore"];; (* LISTA DI STRINGHE *)
- : string list = ["casa"; "famiglia"; "amore"]
# [9; 16; 25];; (* LISTA DI INTERI *)
- : int list = [9; 16; 25]
```

Il simbolo `[]` indica una lista vuota, mentre l'operatore *cons*, rappresentato con `::`, permette di aggiungere un elemento in testa ad una lista.

```
# [];; (* LISTA VUOTA *)
- : 'a list = []
# 3::[];; (* AGGIUNGE UN INT ALLA LISTA VUOTA *)
- : int list = [3]
# "rosso"::["verde";"blu"];; (* AGGIUNGE UNA STRINGA IN TESTA ALLA LISTA *)
```

```
- : string list = ["rosso"; "verde"; "blu"]
```

L'operatore `cons` è associativo a destra, quindi la scrittura `3::4::5::[]`, interpretabile come `3::(4::(5::[]))`, creerà una lista, inizialmente vuota, a cui verranno aggiunti in testa gli interi `5`, `4` e `3` (nell'ordine dato), ottenendo così la lista `[3; 4; 5]`:

```
# 3::4::5::[];;
- : int list = [3; 4; 5]
# 3::(4::(5::[]));;
- : int list = [3; 4; 5]
```

Due liste sono uguali se hanno la stessa lunghezza e se gli elementi corrispondenti sono uguali:

```
# [1;2;3] = 1::2::3::[];;
- : bool = true
# [[1];[2;3]] = [4/5+1]::[[2;6-3]];; (* LISTE DI LISTE DI INT *)
- : bool = true
```

**1.6.9. I record.** Un record è costituito da un numero finito di *campi*, di tipo non necessariamente omogeneo, ciascuno dei quali è identificato da un'*etichetta* e contiene un valore di un tipo determinato. A differenza delle tuple, le componenti di un record (i campi) sono distinti mediante i nomi (le etichette), anziché per la posizione.

In Ocaml, prima di utilizzare espressioni di tipo record, è necessario definire il particolare tipo di record che si vuole utilizzare, allo scopo di introdurre le etichette dei campi.

```
# (* DEFINIZIONE TIPO DI RECORD *)
  type studente = {nome: string; matricola: int * int};;
type studente = { nome : string; matricola : int * int; }
# (* DICHIARAZIONE DI UN RECORD *)
  let s1 = {nome = "pippo"; matricola = 123456, 12};;
val s1 : studente = {nome = "pippo"; matricola = (123456, 12)}
# (* SELEZIONE DEI SINGOLI CAMPI DI UN RECORD *)
  s1.nome;;
- : string = "pippo"
# s1.matricola;;
- : int * int = (123456, 12)
```

## 1.7. Tail Recursion (ricorsione in coda)

In linea di principio, una chiamata ricorsiva non opportunamente ottimizzata ha bisogno di allocare spazio nello stack (a tempo di esecuzione) per ogni chiamata che non è ancora terminata: questo spreco di memoria la rende inefficiente per rappresentare i cicli. *La ricorsione in coda è invece equivalente ai costrutti iterativi e altrettanto efficiente*.

Si ha *tail recursion* quando la chiamata ricorsiva è l'ultima istruzione eseguita prima di terminare la funzione. Gli stati intermedi necessari prima di arrivare al "caso base" *(vedi 1.5.3)* non saranno memorizzati nello stack, bensì passati all'interno della chiamata ricorsiva. A tale scopo, di solito viene utilizzata un'opportuna variabile, detta *accumulatore* (parametro che accumula il risultato parziale). *Anche se formalmente ricorsiva, la tail recursion dà luogo ad un processo computazionale di tipo iterativo.*

ALGORITHM 1.7.1. *Fattoriale con ricorsione generica*

```
# let rec fatt n =
      if n=0 then 1
      else n*fatt(n-1);;
val fatt : int -> int = <fun>
# fatt(4);;
- : int = 24
```

Il flusso di esecuzione dell'algoritmo sarà:

```
fatt(4) --> 4*fatt(3) --> 4*3*fatt(2) --> 4*3*2*fatt(1) -->
4*3*2*1*fatt(0) --> 4*3*2*1*1 = 24
```

ALGORITHM 1.7.2. *Fattoriale tail-recursive (funzione ausiliaria più funzione principale per inizializzare l'ausiliaria)*

```
# let rec aux_fatt (n, acc) =
      if n=0 then acc
      else aux_fatt (n-1, acc*n);;
val aux_fatt : int * int -> int = <fun>
# let fatt'(n) = aux_fatt (n,1);;
val fatt' : int -> int = <fun>
# fatt'(4);;
- : int = 24
```

ALGORITHM 1.7.3. *Fattoriale tail-recursive, equivalente al precedente (funzione ausiliaria dichiarata localmente all'interno della principale)*

```
# let fatt" n =
      let rec aux_fatt (n, acc) =
          if n=0 then acc
          else aux_fatt (n-1, acc*n)
      in aux_fatt(n,1);;
val fatt" : int -> int = <fun>
# fatt"(4);;
- : int = 24
```

Il flusso di esecuzione di entrambi gli algoritmi tail-recursive sarà:

```
fatt'(4) --> aux_fatt(4,1) --> aux_fatt(3,4) --> aux_fatt(2,12) -->
aux_fatt(1, 24) --> aux_fatt(0, 24) --> 24
```

ALGORITHM 1.7.4. *Potenza con ricorsione generica*

```
# let rec pot (base, esp) =
      if esp=0 then 1
      else base*pot(base, esp-1);;
val pot : int * int -> int = <fun>
# pot (2,3);;
- : int = 8
```

ALGORITHM 1.7.5. *Potenza tail-recursive*

```
# let pot' (base, esp) =
      let rec aux_pot (base, esp, acc) =
          if esp=0 then acc
          else aux_pot (base, esp-1, acc*base)
      in aux_pot (base, esp, 1);;
```

```
val pot' : int * int -> int = <fun>
# pot' (2,3);;
- : int = 8
```

## 1.8. Pattern Matching

Molti linguaggi di programmazione prevedono un costrutto che permette di indirizzare il flusso di esecuzione del programma in base al valore di un'espressione. O'Caml fornisce un costrutto, noto come Pattern Matching, che permette il confronto tra un pattern (schema), definito all'interno della funzione, con l'argomento con cui si richiama la funzione.

Il pattern matching confronta il valore passato come argomento con il primo possibile pattern; se fallisce, allora passa al pattern successivo. E' quindi importante l'ordine con cui viene scritto il pattern.

La variabile dummy (cioè "fittizia", "di comodo") underscore "_" indica qualunque possibile valore attribuibile alla variabile.

E' possibile scegliere una delle seguenti sintassi:

```
let [rec] nome_funzione = function
    pattern_1 -> espressione_1
    | ...
    | pattern_n -> espressione_n
```

oppure:

```
let [rec] nome_funzione argomento = match argomento with
    pattern_1 -> espressione_1
    | ...
    | pattern_n -> espressione_n
```

*Se l'argomento è una tupla, è possibile indicare in* `match argomento with` *un sottoinsieme dei suoi parametri.*

ALGORITHM 1.8.1. *Fattoriale con pattern matching*

```
# let rec fatt n = match n with
      0 -> 1
      | n -> n * fatt (n-1);;
val fatt : int -> int = <fun>
# fatt (4);;
- : int = 24
```

ALGORITHM 1.8.2. *Fattoriale con pattern matching e variabile dummy underscore*

```
# let rec fatt' n = match n with
      0 -> 1
      | _ -> n * fatt' (n-1);;
val fatt' : int -> int = <fun>
# fatt' (4);;
- : int = 24
```

ALGORITHM 1.8.3. *Fattoriale con pattern matching e omissione degli argomenti, per mezzo della parola chiave "function"*

```
# let rec fatt = function
      0 -> 1
      | n -> n * fatt (n-1);;
val fatt : int -> int = <fun>
# fatt (4);;
- : int = 24
```

ALGORITHM 1.8.4. *Potenza con pattern matching e parola chiave function*

```
# let rec pot = function
      (_,0) -> 1
      | (b,e) -> b*pot(b,e-1);;
val pot : int * int -> int = <fun>
# pot (2,3);;
- : int = 8
```

ALGORITHM 1.8.5. *Serie di Fibonacci (http://it.wikipedia.org/wiki/Numeri_di_Fibonacci) con pattern matching e parola chiave function*

```
# let rec fib = function
      0 -> 0
      | 1 -> 1
      | n -> fib (n-1) + fib (n-2);;
val fib : int -> int = <fun>
# fib (2);;
- : int = 1
# fib (3);;
- : int = 2
# fib (4);;
- : int = 3
# fib (5);;
- : int = 5
# fib (6);;
- : int = 8
```

ALGORITHM 1.8.6. *Implementazione porta AND*

```
# let and_port = function
      (true,e) -> e
      | (_,_) -> false;;
val and_port : bool * bool -> bool = <fun>
# and_port (true,true);;
- : bool = true
# and_port (false,true);;
- : bool = false
# and_port (true, false);;
- : bool = false
# and_port (false, false);;
- : bool = false
```

ALGORITHM 1.8.7. *Implementazione porta OR*

```
# let or_port = function
      (false, false) -> false
      | (_,_) -> true;;
val or_port : bool * bool -> bool = <fun>
# or_port (true, false);;
- : bool = true
```

## 1.9. Le eccezioni

Se una funzione non è definita per tutti i suoi argomenti, come nel caso del fattoriale, è opportuno fare in modo che, se richiamata con valori al di fuori del suo dominio, segnali un errore, piuttosto che causare un comportamento non prevedibile o addirittura generare un loop infinito. A questo scopo, Ocaml dispone di un tipo particolare di dati, le eccezioni, che possono essere usate come argomento o valore di qualsiasi funzione. Nel modulo Pervasives sono definite alcune eccezioni di base, come "Match failure" oppure "Division by zero", che possono essere estese dal programmatore, mediante dichiarazioni nella forma:

```
exception NomeEccezione
```

Per far sì che una funzione riporti un'eccezione, occorre "sollevare" *(to raise)* l'eccezione, mediante la parola chiave `raise`. Nel codice seguente, la funzione fattoriale, definita in due modi diversi per mostrare il diverso comportamento del programma in seguito al medesimo input, è richiamata con un argomento negativo:

```
# (* Eccezione non dichiarata -> stack overflow *)
  let rec fatt = function
      0 -> 1
      | n -> n * fatt (n-1);;
val fatt : int -> int = <fun>
# fatt (-1);;
Stack overflow during evaluation (looping recursion?).

# (* Eccezione dichiarata -> la funzione termina *)
# exception Neg_argument;;
exception Neg_argument
# let fatt n =
      let rec p_fatt = function
          0 -> 1
          | n -> n * p_fatt (n-1)
      in if n>=0
          then p_fatt n
          else raise Neg_argument;;
val fatt : int -> int = <fun>
# fatt(-1);;
Exception: Neg_argument.
```

**1.9.1. Propagazione automatica e "cattura" delle eccezioni.** La valutazione di un'espressione viene immediatamente interrotta se al suo interno viene sollevata un'eccezione, come nel caso seguente:

```
# 4 * fatt(-5) + 1;;
Exception: Neg_argument.
```

Tale comportamento è noto come *propagazione automatica delle eccezioni* e può essere evitato tramite un *exception handler* (cattura di un'eccezione) specificando che, se una determinata sottoespressione ha un valore "normale" allora si riporta tale valore, altrimenti, se la valutazione solleva un'eccezione, allora si deve riportare un altro valore. La sintassi è:

```
try Espressione_1 with Exception -> Espressione_2
```

Le due espressioni devono essere dello stesso tipo: se la prima solleva l'eccezione specificata con il `with`, allora viene riportato il valore della seconda espressione, altrimenti quello della prima, a condizione che non venga sollevata un'eccezione diversa da quella specificata. Infatti si ha:

```
# let twofatt (n, m) =
      try fatt(n) * fatt (m) with Neg_argument -> 0;;
val twofatt : int * int -> int = <fun>
# twofatt (2, -3);;
- : int = 0
# twofatt (2, 3);;
- : int = 12
```

## 1.10. Riepilogo: il nucleo di un linguaggio funzionale

**Tecniche per risolvere problemi: riduzione a sottoproblemi più semplici**
*http://logica.uniroma3.it/csginfo/fondamenti/cialdea/slides/04.pdf*

**Esercizi di riepilogo**
*http://logica.uniroma3.it/csginfo/fondamenti/cialdea/slides/esercizi-3-4.pdf*

## 1.11. Esercizi svolti

EXERCISE 1.12. Definire una funzione `second` che, applicata ad una coppia, ne restituisca il secondo elemento.

```
# let second (n, m) = m;;
val second : 'a * 'b -> 'b = <fun>
# second (5, "a");;
- : string = "a"
```

EXERCISE 1.13. Definire ricorsivamente una funzione che, applicata ad un intero positivo `n`, determini se `n` è potenza di 2 (la funzione riporterà un booleano).

```
# let rec potenza2 = function
      1 -> true
      | n -> ((n mod 2) = 0) && potenza2 (n/2);;[4]
val potenza2 : int -> bool = <fun>
# potenza2 (8);;
- : bool = true
```

EXERCISE 1.14. Definire una funzione `sum` che, applicata a due interi positivi, ne determini la somma con ricorsione generica e con ricorsione in coda.

```
# let rec sum = function
      (n, 0) -> n
      | (n, m) -> succ (sum(n,pred(m)));;
val sum : int * int -> int = <fun>
# sum (12, 5);;
- : int = 17
```

[4] ***In questo caso, Ocaml ha un approccio lazy* (pigro) *alla valutazione dell'espressione, nel senso che ne valuta il secondo elemento, cioè la chiamata ricorsiva, solo se ce n'è realmente bisogno.***

```
# let rec sum (a, b) = match a with
      0 -> b
      | _ -> sum (a-1, b+1);;
val sum : int * int -> int = <fun>
# sum (3, 45);;
- : int = 48
```

EXERCISE 1.15. Definire una funzione `product` che, applicata a due interi positivi, ne determini il prodotto con ricorsione in coda.

```
# let product (a,b) =
        match (a,b) with
         (0,_) | (_,0) -> 0
         | (1,_) -> b
         | (_,1) -> a
         | _ ->
        let rec aux_product (m, n, acc) = match n with
                | 0 -> acc
                | _ -> aux_product (m, n-1, acc+m)
        in aux_product (a, b, 0);;
val product : int * int -> int = <fun>
# product (4, 6);;
- : int = 24
```

EXERCISE 1.16. Scrivere una funzione che, applicata ad un intero $d$ e una stringa $m$, ritorti *true* se e solo se *(d, m)* rappresenta una data corretta (ad esempio `(28, "febbraio")`) - assumendo che l'anno non sia bisestile.

*« 30 dì conta Novembre, con April, Giugno e Settembre, di 28 ce n'è uno, tutti gli altri ne han 31... »*

```
# let date (g, m) = match m with
      "gennaio" | "marzo" | "maggio" | "luglio" | "agosto"
      | "ottobre" | "dicembre"  -> g>=1 && g<=31
      | "febbraio" -> g>=1 && g<=28
      | "aprile" | "giugno"
      | "settembre" | "novembre"-> g>=1 && g<=30
      | _ -> false;;
val date : int * string -> bool = <fun>
# date (28, "febbraio");;
- : bool = true
# date (2, "gennaio");;
- : bool = true
# date (31, "novembre");;
- : bool = false
```

EXERCISE 1.17. Si consideri il problema di determinare il numero di divisori di un intero positivo $n$: si possono scandire i numeri da 1 a $n$ e, per ciascuno di essi, si determina se divide $n$; in caso positivo, si incrementa un contatore (inizializzato a 0). Si implementi questo algoritmo, utilizzando una funzione ausiliaria che svolga il ruolo dell'iterazione.

Soluzione 1:

```
# let rec f (n, div, acc) =
      if (div > n) then acc
      else if ((n mod div) = 0) then f (n, div+1, acc+1)
      else f (n, div+1, acc);;
```

```
val f : int * int * int -> int = <fun>
# let divisori n = f (n, 1, 0);;
val divisori : int -> int = <fun>
# divisori (10);;
- : int = 4
```

Soluzione 2:

```
# let rec aux = function
      (_,1,count) -> count +1
      | (n, k, count) ->
          if n mod k = 0
          then aux (n, k-1, count+1)
          else aux (n, k-1, count);;
val aux : int * int * int -> int = <fun>
# let divisori (n) = aux (n, n, 0);;
val divisori : int -> int = <fun>
# divisori (10);;
- : int = 4
```

EXERCISE 1.18. Si consideri il problema dell'esercizio precedente. In questo caso si vuol stampare la lista dei divisori.

```
# let rec f (n, div, acc) =
      if (div > n) then acc
      else if ((n mod div) = 0)
           then f (n, div+1, acc^string_of_int(div)^" ")
      else f (n, div+1, acc);;
val f : int * int * string -> string = <fun>
# let divisori n = f (n, 1, "");;
val divisori : int -> string = <fun>
# divisori (20);;
- : string = "1 2 4 5 10 20 "
```

EXERCISE 1.19. Implementare la sommatoria $\sum_{k=n}^{m} = f(k)$

```
# let rec sum (f, n, m) =
      if n > m then 0
      else f(n) + sum (f, n+1, m);;
val sum : (int -> int) * int * int -> int = <fun>

# sum((function(x)->x*x),1,5);;
- : int = 55
```

EXAMPLE. Scrivere una funzione che calcoli il valore minimo di una funzione `f: int -> int` in un intervallo finito.

```
# exception NoIntervallo;;
exception NoIntervallo

# let rec minimo (f, sx, dx) =
        if (sx > dx) then raise NoIntervallo
        else if (sx = dx) then f(dx)
        else min (f(sx)) (minimo(f, sx+1, dx));;
val minimo : (int -> 'a) * int * int -> 'a = <fun>
```

```
# minimo ((function a -> a*100), 1, 10);;
- : int = 100
# minimo ((function a -> 100/a), 1, 10);;
- : int = 10
# minimo ((function a -> 5+a), 1, 1);;
- : int = 6
# minimo ((function a -> 5+a), 2, 1);;
Exception: NoIntervallo.
```

CAPITOLO 2

# Le liste

Se a, b, c, ... sono elementi di uno stesso tipo $\alpha$, allora una lista di elementi di tipo $\alpha$ è una sequenza finita di tali elementi. Le liste vengono denotate in OCaml racchiudendo gli elementi fra parentesi quadre e separandoli mediante punti e virgola. Ad esempio, [3;6;3;10] è una lista di interi.

Ciascuna lista si può vedere costruita a partire dalla lista vuota (indicata con []), mediante l'operazione di "inserimento in testa", indicata con :: e chiamata operazione "cons". Ad esempio, la lista [3;6;3;10] si ottiene inserendo 3 in testa alla lista [6;3;10], ovvero 3::[6;3;10] = [3;6;3;10]. Di fatto, [3;6;3;10] è un modo compatto di scrivere l'espressione 3::(6::(3::(10::[]))).

In generale, l'insieme delle liste di elementi di un tipo $\alpha$ costituisce il tipo $\alpha$ list. Ad esempio, [3;6;3;10] è un'espressione di tipo int list, e [[4;5];[6;7;8];[9;10]] è una int list list, cioè una lista di liste di interi.

Si ricordi che la virgola viene utilizzata per le tuple, il punto e virgola per le liste, infatti si ha:

```
# (* lista di coppie i cui elementi sono liste di interi *)
  [[1;2],[3;4;5]];;
- : (int list * int list) list = [([1; 2], [3; 4; 5])]
# (* lista di liste di interi *)
  [[1;2];[3;4;5]];;
- : int list list = [[1; 2]; [3; 4; 5]]
```

L'insieme delle liste di tipo $\alpha$ list può essere definito come segue:

(1) La lista vuota [] è una $\alpha$ list.
(2) Se x è di tipo $\alpha$ e xs è una $\alpha$ list, allora (x::xs) è una $\alpha$ list.
(3) nient'altro è una $\alpha$ list.

La lista vuota e l'operatore cons sono quindi i *costruttori* del tipo lista. Qualsiasi lista ha una delle due forme seguenti: [] oppure $\alpha$ list.

L'append, indicato con il simbolo @, è l'operatore di concatenazione:

```
# [1;2] @ [3;4;5];;
- : int list = [1; 2; 3; 4; 5]
```

TABELLA 1. Pattern Matching con le liste

| *pattern* | *espressione da confrontare* | *espressione equivalente* | *esito confronto e legami* |
|---|---|---|---|
| **[]** | **[]** | | **successo** |
| **[]** | **[1]** | | **fallimento** |
| **[]** | **[1;2]** | | **fallimento** |
| **[x]** | **[]** | | **fallimento** |
| **[x]** | **[1]** | | **x=1** |
| **[x]** | **[1;2]** | | **fallimento** |
| **x::[]** | **[]** | | **fallimento** |
| **x::[]** | **[1]** | **1::[]** | **x=1** |
| **x::[]** | **[1;2]** | **1::[2]** | **fallimento** |
| **x::y::[]** | **[]** | | **fallimento** |
| **x::y::[]** | **[1]** | **1::[]** | **fallimento** |
| **x::y::[]** | **[1;2]** | **1::2::[]** | **x=1, y=2** |
| **x::y** | **[]** | | **fallimento** |
| **x::y** | **[1]** | **1::[]** | **x=1, y=[]** |
| **x::y** | **[1;2]** | **1::[2]** | **x=1, y=[2]** |
| **x::y** | **[1;2;3]** | **1::[2;3]** | **x=1, y=[2;3]** |
| **x::xs** | **[]** | | **fallimento** |
| **x::xs** | **[1]** | **1::[]** | **x=1, xs=[]** |
| **x::xs** | **[1;2]** | **1::[2]** | **x=1, xs=[2]** |
| **x::xs** | **[1;2;3]** | **1::[2;3]** | **x=1, xs=[2;3]** |
| **x::y::xs** | **[]** | | **fallimento** |
| **x::y::xs** | **[1]** | **1::[]** | **fallimento** |
| **x::y::xs** | **[1;2]** | **1::2::[]** | **x=1, y=2, xs=[]** |
| **x::y::xs** | **[1;2;3]** | **1::2::[3]** | **x=1, y=2, xs=[3]** |
| **x::y::xs** | **[1;2;3;4;5]** | **1::2::[3;4;5]** | **x=1, y=2. xs=[3;4;5]** |

### 2.1. Pattern Matching con le liste

Nel definire funzioni sulle liste è possibile utilizzare il pattern matching. La tabella 1 , tratta da [**5**], riporta, in ogni riga, un pattern di esempio, un'espressione con il cui il pattern viene confrontato, il modo di leggere l'espressione applicando l'operatore `::` nella stessa maniera in cui è specificato nel pattern e, infine, l'esito del confronto, con gli eventuali legami stabiliti dall'operazione di pattern matching.

Affinché il pattern matching abbia successo, il pattern deve essere scritto nella stessa forma dell'espressione da confrontare o in una equivalente, ottenibile con le regole di composizione delle liste. In particolare, si tenga presente che il pattern matching *ha successo se*:

- ad ogni elemento della lista specificata nel pattern corrisponde uno ed un solo elemento della lista specificata nell'espressione da confrontare;
- nel caso siano presenti nel pattern uno o più `cons`, allora ad ogni lista alla destra di un `cons` corrisponde una lista nell'espressione da confrontare (pattern ed espressione saranno confrontati usando lo stesso numero di `cons`), mentre ad ogni elemento alla sinistra di un `cons` corrisponderà un elemento nell'espressione da confrontare;

- "liste" fanno il match solo con "liste" ed "elementi di liste" solo con "elementi di liste".

Nei pattern è preferibile usare simboli come `xs` o `ys` per indicare le liste (la "s" inglese, usata per indicare il plurale, evidenzia in questo caso una pluralità di elementi).

In un pattern di lista può anche essere presente la variabile dummy (underscore).

## 2.2. Funzioni sulle liste: esempi ed esercizi svolti

Come indicato nel manuale di Ocaml[1], il modulo `List` contiene una serie di funzioni predefinite sulle liste, come ad esempio i due selettori `hd` e `tl` che, applicati ad una lista, ne restituiscono rispettivamente la testa *(cioè il primo elemento)* o la coda *(cioè la lista che si ottiene eliminando la testa)*.

```
# (* Selettore: restituisce la testa della lista *)
  List.hd;;
- : 'a list -> 'a = <fun>

# (* Selettore: restituisce la coda della lista *)
  List.tl;;
- : 'a list -> 'a list = <fun>

# List.hd [1;2;3];;
- : int = 1

# List.tl [1;2;3];;
- : int list = [2; 3]
```

In questa sezione verranno presentate alcune possibili implementazioni di funzioni su liste.

**Head.** Restituisce il primo elemento della lista.

```
# exception EmptyList;;
exception EmptyList

# let hd = function
        x::_ -> x
        | _ -> reise Emptylist;;
val hd : 'a list -> 'a = <fun>
```

**Tail.** Restituisce la coda della lista, cioè la lista che si ottiene eliminando il primo elemento.

```
# let tl = function
  [] -> raise EmptyList
  | _::xs -> xs;;
val tl : 'a list -> 'a list = <fun>
```

[1]http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html

**Null.** Determina se una lista è vuota.

```
# let null lst = lst = [];;
val null : 'a list -> bool = <fun>
```

**Last.** Restituisce l'ultimo elemento della lista

```
# let rec last = function
        [] -> raise Emptylist
        | [x] -> x
        | _::xs -> last xs;;
val last : 'a list -> 'a = <fun>
# last [1;2;3];;
: int = 3
```

**Upto.** Dati due interi, restituisce una lista composta da tutti gli interi compresi tra il primo e il secondo, estremi compresi.

```
# let upto (n, m) =
        let rec aux_upto (n, m, acc) =
                if n>m then acc
                else aux_upto (n, m-1, m::acc)
        in aux_upto (n, m, []);;
val upto : int * int -> int list = <fun>

# upto (0,5);;
- : int list = [0; 1; 2; 3; 4; 5]
```

**Lenght.** Restituisce la lunghezza della lista.

```
# let rec lenght = function
        _::xs -> 1 + lenght xs
        | _ -> 0;;
val lenght : 'a list -> int = <fun>

# lenght [12;1;9];;
- : int = 3
```

**Mem.** Controlla se l'elemento dato fa parte della lista.

```
# let rec mem (n, lst) = match lst with
        [] -> false
        | x::xs -> (n = x) or mem (n, xs);;
  val mem : 'a * 'a list -> bool = <fun>

(* oppure, in versione curryficata: *)
# let rec mem n lst = match lst with
        [] -> false
        | x::xs -> (n = x) or mem n xs;;
val mem : 'a -> 'a list -> bool = <fun>

(* od anche: *)
# let rec mem n = function
        [] -> false
        | x::xs -> (n = x) or mem n xs;;
```

```
val mem : 'a -> 'a list -> bool = <fun>

# mem 8 [1;2];;
- : bool = false
# mem 2 [1;2];;
- : bool = true
```

**Init.** Restituisce la lista che si ottiene eliminando l'ultimo elemento

```
# let rec init = function
        [] -> raise EmptyList
        | [x] -> []
        | x::xs -> x::init(xs);;
val init : 'a list -> 'a list = <fun>
# init [3;4;5];;
- : int list = [3; 4]
```

**Append.** Implementare l'append senza usare l'operatore di concatenazione @.

```
# let rec append = function
        ([],ys) -> ys
        | (x::xs,ys) -> x::append(xs,ys);;
val append : 'a list * 'a list -> 'a list = <fun>

(* oppure *)

# let rec append (xs, ys) = match xs with
        [] -> ys
        | z::zs -> z::append(zs,ys);;
val append : 'a list * 'a list -> 'a list = <fun>

(* oppure, in versione curryficata: *)

# let rec cappend xs ys = match xs with
        [] -> ys
        | z::zs -> z::cappend zs ys;;
val cappend : 'a list -> 'a list -> 'a list = <fun>

# cappend [1;3] [7];;
- : int list = [1; 3; 7]
```

**Maxinlist.** Restituisce l'elemento maggiore fra tutti quelli della lista.

```
# let rec maxinlist = function
        [a] -> a
        | [] -> raise EmptyList
        | a::b::cs -> maxinlist ((max a b)::cs);;
        val maxinlist : 'a list -> 'a = <fun>

# maxinlist [3;1;7;4];;
- : int = 7

let rec maxinlist = function
        [] -> raise EmptyList
```

```
        | [x] -> x
        | x::xs -> max x (maxinlist xs);;
```

**Reverse.** Restituisce una lista con gli elementi invertiti.

```
# let rec reverse = function
        [] -> []
        | x::xs -> (reverse xs) @ [x];;
val reverse : 'a list -> 'a list = <fun>

# reverse [1;2;3];;
- : int list = [3; 2; 1]
```

Si noti che, in questo caso, le parentesi tonde per racchiudere `(reverse xs)` non sono necessarie, perché la chiamata di funzioni ha la precedenza sugli operatori. Ad esempio, `funzione n-1` equivale a `(funzione n)-1`.

Versione equivalente con ricorsione in coda:

```
#let tr_reverse lista =
        let rec aux invertita = function
                [] -> invertita
                | x::xs -> aux (x::invertita) xs
        in aux [] lista;;
val tr_reverse : 'a list -> 'a list = <fun>

# tr_reverse [1;2;3;4];;
- : int list = [4; 3; 2; 1]
```

In questo caso, invece, sono necessarie le parentesi per indicare la precedenza del `cons` in `(x::invertita)`. La funzione ausiliaria locale è sia tail-recursive sia curryficata: il primo argomento viene associato alla variabile `invertita`, mentre il secondo viene utilizzato per il pattern matching.

**Take.** Definire una funzione `take: int * 'a list -> 'a list`, tale che restituisca una lista con i primi $n$ elementi della lista passata come argomento.

```
let rec take = function
        (0,_) -> []
        | (_,[]) -> []
        | (n, x::xs) -> x::take(n-1,xs);;
val take : int * 'a list -> 'a list = <fun>

# take (2,[1;3;5;7]);;
- : int list = [1; 3]
```

**Zip.** Data la coppia di liste `([x1,x2,x3,...,xn];[y1,y2,...,yn])` restituisce la lista delle coppie `[(x1,y1),(x2,y2),...,(xn,yn)]`

```
# exception LunghezzaDiversa;;
# let rec zip = function
        ([],[]) -> []
        | (_,[]) | ([],_) -> raise LunghezzaDiversa
        | (x::xs, y::ys) -> (x,y)::zip(xs,ys);;
val zip : 'a list * 'b list -> ('a * 'b) list = <fun>
```

```
# zip ([1;2;3],[4;5;6]);;
- : (int * int) list = [(1, 4); (2, 5); (3, 6)]
```

**Unzip.** Inversa alla precedente: data una lista di coppie, restituisce una coppia di liste.

```
let rec unzip = function
        [] -> ([],[])
        | (x,y)::ps ->  let (f,s) = unzip ps
                            in (x::f,y::s);;
val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>

# unzip [(1,2);(3,4);(5,6)];;
- : int list * int list = ([1; 3; 5], [2; 4; 6])
```

**Copy.** Scrivere una funzione `copy: int * 'a -> 'a list` che, applicata ad una coppia `(n,x)`, restituisca la lista di lunghezza `n` i cui elementi sono tutti uguali a `x`.

```
# exception NumNegativo;;
exception NumNegativo

# let rec copy = function
        (0,_) -> []
        |  (n,x) -> if n<0 then raise NumNegativo else x :: copy (n-1, x);;
val copy : int * 'a -> 'a list = <fun>

# copy (3,'R');;
- : char list = ['R'; 'R'; 'R']
```

**Nondec.** Scrivere un predicato `nondec: int list -> bool` che, applicato ad una lista `lst`, restituisca `true` se gli elementi di `lst` sono in ordine non decrescente, `false` altrimenti.

```
# let rec nondec = function
        [] -> true
        | [x] -> true
        | x::y::xs -> if y<x then false else nondec (y::xs);;
val nondec : 'a list -> bool = <fun>

# nondec [3;4;4;5];;
- : bool = true
```

**Duplica.** Scrivere una funzione che, applicata ad una lista `xs=[x1;x2;...;xn]`, duplichi ogni elemento della lista, cioè restituisca `[x1;x1;x2;x2;...;xn;xn]`.

```
# let rec duplica = function
        [] -> []
        | x::xs -> x::x::duplica(xs);;
val duplica : 'a list -> 'a list = <fun>

# duplica [1;2;3];;
- : int list = [1; 1; 2; 2; 3; 3]
```

**Pairwith.** Definire una funzione `pairwith` che, applicata ad un oggetto `y` e una lista `xs=[x1;x2;...;xn]`, restituisca la lista `[(y,x1);(y,x2);...;(y,nx)]`. Determinare il tipo della funzione.

```
let rec pairwith y = function
        [] -> []
        | x::xs -> (y,x)::(pairwith y xs);;
val pairwith : 'a -> 'b list -> ('a * 'b) list = <fun>

# pairwith 'k' [3;4;5];;
- : (char * int) list = [('k', 3); ('k', 4); ('k', 5)]
```

**Map.** Scrivere una funzione che applicata ad una lista `lst=[x0;x1;x2;...;xk]` restituisca la lista di coppie `[(0,x0);(1;x1);(2;x2);...;(k,xk)]`. Determinare il tipo della funzione.

```
let map xs =
        let rec aux = function
                ([],_) -> []
                | (x::xs, n) -> (x,n)::aux(xs,n+1)
        in aux (xs,0);;
val map : 'a list -> ('a * int) list = <fun>

# map ['a';'b';'c'];;
- : (char * int) list = [('a', 0); ('b', 1); ('c', 2)]
```

**Position.** Scrivere una funzione `position: 'a list * 'a -> int` tale che `position (lst,x)` restituisca la posizione della prima occorrenza di `x` in `lst` (contando a partire da 0) se `x` occorre in `lst`, oppure sollevi altrimenti un'eccezione.

```
# exception InsiemeVuoto;;
exception InsiemeVuoto

# let position (xs,m) =
        let rec aux = function
                (_,[]) -> raise InsiemeVuoto
                | (p,x::xs) -> if x=m then p else aux (p+1,xs)
        in aux (0,xs);;
val position : 'a list * 'a -> int = <fun>

# position (['c';'a';'s';'a'],'a');;
- : int = 1
```

**Nth.** Scrivere una funzione `nth: ('a list * int) -> 'a` che, applicata ad una lista `lst` di lunghezza *n* e a un numero *k* compreso tra 0 e *n*-1, restituisca il *k*-esimo elemento di lst.

```
# exception ListaVuota;;
exception ListaVuota

# let rec nth = function
        [],_ -> raise ListaVuota
        | x::xs,0 -> x
        | x::xs,k -> nth (xs,k-1);;
```

```
val nth : 'a list * int -> 'a = <fun>

# nth (['c';'a';'s';'a'],2);;
- : char = 's'
```

**Alternate.** Scrivere una funzione `alternate: 'a list -> 'a list` che, applicata ad una lista `lst`, restituisca la lista contenente tutti e solo gli elementi di `lst` che si trovano in posizione dispari. Per convenzione, il primo elemento di una lista si trova in posizione 0.

```
let rec alternate = function
        [] -> []
        | [x] -> []
        | x::y::ys -> y :: alternate ys;;
val alternate : 'a list -> 'a list = <fun>

# alternate [0;1;2;3;4;5];;
- : int list = [1; 3; 5]
```

**Minmax.** Scrivere un programma che, data una lista di liste di interi, restituisca il valore minimo tra i massimi di ciascuna lista.

```
# exception ListaVuota;;
exception ListaVuota
# let rec massimo = function
        [] -> raise ListaVuota
        | [x] -> x
        | x::y::ys -> if (x>y) then massimo (x::ys) else massimo (y::ys);;
val massimo : 'a list -> 'a = <fun>
# let rec minimo = function
        [] -> raise ListaVuota
        | [x] -> x
        | x::y::ys -> if (x<y) then minimo (x::ys) else minimo (y::ys);;
val minimo : 'a list -> 'a = <fun>
# let rec maxmin = function
        [] -> raise ListaVuota
        | [x] -> massimo x
        | x::y::ys -> minimo [(massimo x);(maxmin (y::ys))];;
val maxmin : 'a list list -> 'a = <fun>
# maxmin [[3;100;1;9];[2;10;20];[80;65;4]];;
- : int = 20
```

## 2.3. Funzioni di ordine superiore sulle liste

Le funzioni di ordine superiore possono prendere funzioni come argomenti e/o restituirle come output. Molte di queste sono definite nel modulo List.

Nelle seguenti funzioni verrà fatto uso di predicati, cioè di funzioni del tipo: `'a -> bool`.

**2.3.1. Map: `('a -> 'b) -> 'a list -> 'b list`.** Data una funzione e una lista, restituisce la lista che si ottiene applicando la funzione ad ogni elemento della lista.

```
# let rec map f = function
        [] -> []
        | x::xs -> f x :: map f xs;;
map : ('a -> 'b) -> 'a list -> 'b list
# let double x = 2 * x;;
val double : int -> int = <fun>
# map double;;
- : int list -> int list = <fun>
# map double [[4;5];[5]];;
This expression has type 'a list but is here used with type int
# map (map double);;
- : int list list -> int list list = <fun>
# map (map double) [[4;5];[5]];;
int list list = [[8; 10]; [10]]
```

**2.3.2. Inits: `'a list -> 'a list list`.** Data una lista, restituisce una lista di liste, nella forma:

```
inits [1;2;3] -> [[1];[1;2];[1;2;3]]
```

L'implementazione si avvale delle funzioni Map (precedentemente analizzata) e Cons:

```
# let cons x lst = x::lst;;
val cons : 'a -> 'a list -> 'a list = <fun>
# let rec inits = function
        [] -> []
        | x::xs -> [x] :: List.map (cons x) (inits xs);;
val inits : 'a list -> 'a list list = <fun>
# inits [1;2;3];;
- : int list list = [[1]; [1; 2]; [1; 2; 3]]
```

Infatti si ha:

```
inits [1;2;3] ->
[1] :: List.map (cons 1) (inits [2;3]) ->
[1] :: List.map (cons 1) ([2] :: List.map (cons 2) (inits [3])) ->
[1] :: List.map (cons 1) ([2] :: List.map (cons 2) ([3] :: List.map (cons 3) (i
[1] :: List.map (cons 1) ([2] :: List.map (cons 2) ([3] :: List.map (cons 3) []
[1] :: List.map (cons 1) ([2] :: List.map (cons 2) ([3] :: [])) ->
[1] :: List.map (cons 1) ([2] :: List.map (cons 2) ([[3]])) ->
[1] :: List.map (cons 1) ([2] :: [[2;3]]) ->
[1] :: List.map (cons 1) ([[2];[2;3]]) ->
[1] :: [[1;2];[1;2;3]] ->
[[1];[1;2];[1;2;3]]
```

# Bibliografia


[1] **The Caml Language**
*http://caml.inria.fr/*

[2] **The Objective Caml system - Documentation and user's manual**
*http://caml.inria.fr/pub/docs/manual-ocaml/*

[3] **Objective CAML Tutorial**
*http://www.ocaml-tutorial.org/*

[4] **Using Ocaml on Linux**
*http://www.cis.ksu.edu/~ab/Courses/505/spr06/docs/ocaml.pdf*

[5] **Introduzione alla Programmazione Funzionale**
*M. Cialdea Mayer e C. Limongelli*
*ed. Esculapio, 2005*

[6] **Ezine di programmazione funzionale**
*http://xoomer.virgilio.it/ubrcianc/EPF/numero_1/numero_1.pdf*

[7] **Linguaggi formali e compilatori**
*http://sra.itc.it/people/zini/Didattica/2004-2005/esercitazione1.pdf*

[8] **Programmazione Funzionale**
*http://www.dia.uniroma3.it/~lambda/pf/materiale/slides/02-ocaml.pdf*

[9] **Tecniche per risolvere problemi: riduzione a sottoproblemi più semplici**
*http://logica.uniroma3.it/csginfo/fondamenti/cialdea/slides/04.pdf*

[10] **Introduction to the Objective Caml Programming Language**
*http://www.cis.ksu.edu/~ab/Courses/505/spr05/ocaml-book.pdf*

[11] **Objective Caml**
*http://it.wikipedia.org/wiki/Objective_Caml*